# Using Content Addressing to Transfer Virtual Machine State

Niraj Tolia, Thomas Bressoud, Michael Kozuch and Mahadev Satyanarayanan

IRP-TR-02-11

Summer 2002



Intel Research

# Using Content Addressing to Transfer Virtual Machine State

Niraj Tolia[†‡], Thomas Bressoud[‡⋆], Michael Kozuch[‡], and Mahadev Satyanarayanan[†‡]
[‡]Intel Research Pittsburgh, [†]Carnegie Mellon University, and [⋆]Denison University

## Abstract

Virtual Machines allow for an easy abstraction as they encapsulate the entire running environment including the state of hardware like the processor. This encapsulation allows for quick suspends and resumes and therefore allows us to explore a number of exciting ideas in the field of user mobility. However one of the disadvantages of this encapsulation is that it can be very large and in the order of a number of gigabytes. Moving such heavy objects across networks can be slow and cumbersome. In this paper, we propose to exploit the presence of predetermined blocks of hashed data at sites closer to the resume site to reconstruct a portion of the Virtual Machine image without having to go back to the distributed file system. We intend to extend the work done for the Virtual Machine based Internet Suspend/Resume [8] system to develop our prototype.

## 1 Introduction

There are several possible scenarios in which user behavior can be anticipated and caches warmed with the VM data to reduce latency. However the scenario when the user arrived unannounced at a particular location is equally valid. The issue becomes even more acute when the suspend and resume site are geographical distant. Suppose John Doe suspends his VM at the Intel Research Lab in Pittsburgh and later wants to resume it while he is waiting for a connecting flight at the Seattle airport. In this scenario the time spent in bringing the system up could be intolerable if a large latency exists between the closest distributed file system (DFS) server where the VM was stored and the resume site at the airport.

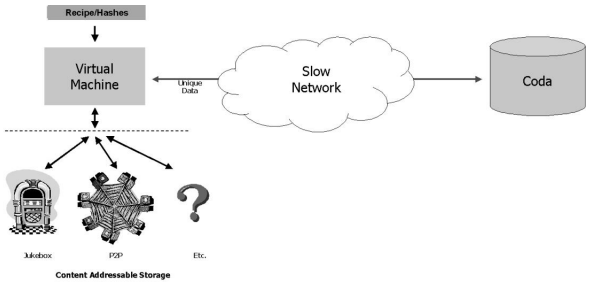


Figure 1: System Diagram

Instead of shipping all the data around, we propose to reconstruct at least a part of the virtual disk at the resume location without having to go out across a low bandwidth network to the DFS server. A pictorial overview of this setup can be seen in Figure 1. The knowledge to reconstruct the disk can be obtained by logically dividing the disk into distinct blocks, generating a unique hash for the data in these blocks and then storing these hashes in a metadata structure. This metadata structure is what we call a recipe. Then, at resume time we initially only need to fetch the recipe. Reconstruction of the disk is done by querying nearby sources for data that match the hashes present in the recipe. The assumption made here is that the bandwidth and latency between the client and nearby sources will be significantly better than what the client sees between it and the DFS server. Possibly sources of data the could be used include content addressable Peer-to-Peer networks like Chord [17] and PAST [5], jukeboxes of software, portable devices, nomadic network storage like Oceanstore [9] or staged storage like the Internet Backplane Protocol (IBP) [13].

This process gives the following series of transfor-

mations that illustrate the actions taken during suspend and resume.

Resume:

$$T_1(RECIPE) \quad \rightarrow \quad DISK$$

Suspend:

$$T_2(DISK) \quad \rightarrow \quad (RECIPE)$$

Update:

$$T_3(\Delta DISK, RECIPE) \quad \rightarrow \quad (RECIPE')$$

The difference between an update and a suspend is that there have only been incremental changes in the disk layout. Thus this does not necessitate doing too much extra work as we can probably reuse a majority of the old $RECIPE$.

We begin by giving a short description of the Internet Suspend/Resume (ISR) project that we are extending in Section 2. We continue in Section 3 with a description of how cryptographic hashes are used and explain our selection of the hash algorithm. The possible options for recipe construction are described in Section 4. The results from our analysis of hard drives to look for commonality is presented in Section 5 while the results from a more weighted approach to look for commonality is presented in in Section 6.

# 2 Internet Suspend/Resume Background

Internet Suspend/Resume is a new capability for mobile computing that mimics the opening and closing of a laptop, but avoids physical transportation of hardware. Through rapid and easy personalization and depersonalization of anonymous hardware, a user is able to effortlessly suspend work at one machine and to resume it at another. ISR can be implemented by layering virtual machine (VM) technology, such as VMWare Workstation [6], on a distributed file system such as NFS, AFS or Coda [7]. The VM performs the encapsulation of execution state and user customization. It saves both the virtual disk and the memory image as files which allow for easier transportation using a distributed file system. This use of a VM for state encapsulation eliminates the need for modifications to applications or the operating system. Thus ISR's support for operating systems is limited only by the operating systems supported by the VM.

# 3 Hashing

As described earlier, we want to construct a recipe for a disk using hashes. It is essential that a collision resistant hash algorithm be selected to decompose and reconstruct a disk. Apart from the obvious problem of incorrect disk reconstruction, the possibility of collision opens up opportunities for malicious users to substitute incorrect data and significantly destabilize the Virtual Machine. Strong hashing algorithms prevent this and their self validating nature allows us to get data from untrusted peers or servers.

While a number of strong cryptographic hashing algorithms exist, we use the SHA-1 [12] algorithm for our prototype. SHA-1 is the second version of the Secure Hash Algorithm that was developed by National Institute of Standards and Technology (NIST). It produces a 160 bit (20 byte) hash from the input data that is more resistant to collisions and brute force and inversion attacks than, say, MD5 [16] which has a 128 bit digest. Another option available is to use the RIPEMD-160 [4] algorithm that was developed in the European Union. As the name indicates, this cryptographic hash also produces a 160 bit digest and is believed to be collision resistant.

As a side note, MD5 is now designated as a legacy algorithm that should not be used. Work on the algorithm that has shown both pseudo collisions [2] as well as collisions in its compression function [3]. Given the current size of disks today and looking into the near future, SHA-1 and RIPEMD-160 with a 160 bit digest should work well. However with the increase of hard disk capacity by orders of magnitude, hash functions with larger digest sizes would be needed to ensure an absence of collisions.

# 4 Disk Decomposition

A number of options are open on how to construct the recipe that will be used at resume time to reconstruct the disk. This recipe can be easily constructed in the interval time between suspend and resume time. Even if this interval is short, as long as a previous copy of the recipe existed, an incremental update to the recipe would take very little time. It is only the initial cost of building it that is high.

Some of the possible options that can be used for recipe construction are outlined below. We should note that we can attach something similar to a popularity column to all of the following options to keep track of the "hot" blocks obtained from applying algorithms like SEER [10] to the system. This would allow us to intelligently request blocks from nearby sources at the resume time to decrease the total number of misses.

## 4.1 Descriptor Chunks

We can divide the entire disk into fixed sized blocks, say 4 KB, and we will thus have a very simple table of the form shown below.

| Block No | Hash |
|---|---|
| 0 | 160 bit Hash |
| 1 | 160 bit Hash |
| ⋮ | ⋮ |

The 'Block No' column is not necessary in this format but is just provided for clarity. While this is the simplest solution it might be large in terms of size. However an advantage of this is that you might not need to get the entire file but instead demand fetch parts of the file. Another variant of this direct mapping could be to have variable sized blocks with an extra column added for size to the table shown above.

We could also easily organize this into a tree structure to decrease the network costs of fetching the recipe as well as updates which might have to be written out.

## 4.2 Indirect Mapping

The Indirect Mapping scheme is a variant of the above where we try to take advantage of the fact that it is likely that the same block occurs more than once on a disk. Thus we can have a representation for the disk as below.

| Hash | Block List |
|---|---|
| 160 bit Hash | 0, 13, 420 |
| 160 bit Hash | 17, 24, ... |
| ⋮ | ⋮ |

## 4.3 Indirect Mapping with Offset

The Indirect Mapping with Offset scheme uses variable sized blocks to map the disk layout. This is probably going to be very beneficial in the case of computer systems that start with the same build, such as IT systems, but then diverge later as users customize their machines. In this, large signed blocks of "golden build" systems could be stored on network locations near to us but our table would be an offset mapping that would take a chunk with a majority of its contents being unmodified data and then overlay it with a delta. The representation would probably look something like that shown below here.

| Hash | Size | Block | Offset | Data Size |
|---|---|---|---|---|
| 160 bits | 55 MB | 0 | 0 | 53 MB |
| 160 bits | 4 MB | 13,568 | 2 MB | 2 MB |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |

The Size column indicates the size of the block that has the given hash signature while the Block column indicates where on the disk the data we want should be placed. The Offset column indicates how far we need to seek into the block for the data we need and then the Data Size column is referenced to find out how much data after seek offset has to be placed on the virtual disk.

For example, suppose we have a 55 MB block that can be found on a source but our block is different in the last 2 MB. Furthermore the 2 MB block difference can again be found but only as the end of 4 MB chunk. Then the 55 MB block can be fetched and then the 4 MB one. Looking at the offset and the size, we know which part of the second block we need to apply and the block number column would give us the location

on disk where it has to be applied. The table for this is displayed above.

### 4.4 Current Implementation

In our current prototype, we are using the Descriptor Chunks method described in Section 4.1. While the Indirect Mapping in Section 4.3 might sound better, we discovered that a gzip of the file presented by the first method returns much better size savings that an indirect mapping (also gzipped).

### 4.5 Analysis of Data Structures

The size of the disk being used for the current prototypes of ISR is a 4 GB disk with 256 MB RAM. This is the size we are going to assume for the calculations below. Also, as described earlier the hash size for each data chunk is going to be 160 bits (20 bytes).

As described in Section 4.1, we can always break the entire disk up into 4 KB sized chunks. Assuming that the list is sorted by block number, we do not need to record the mapping between hash and block location as it is implicit. Given a flat file and assuming each record to be of 20 bytes (the size of the hash) this will give us a table size of:

$$\frac{4 \times 2^{30}}{4 \times 2^{10}} \times 20 = 20 \times 2^{20} \text{ bytes}$$

As we can see, this gives us a table size of 20 MB which can very clearly be improved upon. While it might not seem to be very large, in the case of larger disks of size 100 GB or more, the table size will balloon to 500 MB or over.

One of the first optimizations that could be made would be to use a tree structure so that it is easy to fault in parts of the recipe on a miss. The presence of locality amongst disk accesses should minimize multiple "recipe-page" misses. This approach would be useful when either a working set policy is used or if the disk is being reconstructed while the VM is running.

Another IT scenario specific solution would to use Variable Size Blocks such as that described in Section 4.3. It is however hard to predict commonality at this point in time with out thorough analysis of disk blocks. However we can make approximations based on certain observations. Let us assume that two machines are running Windows XP with Office XP installed. This typically takes up between 1 to 1.5 Gigabytes on a machine. In case of an IT scenario, where a number of machines have the same build, the set of common data could go up to around 2.5 or even more in special cases.

So in the best case with new machines or almost untouched machines, we can have one chunk of 1.5 Gigabytes that is common between the two and could probably be taken as one chunk. This automatically reduces the recipe size from 20 to 12.5 Megs taking the case where every other block of data has to be mapped as a unique 4K chunk.

## 5 Static Disk Commonality

Before building our initial prototype, we conducted a preliminary survey of machines at our disposal to look at commonality. We have initially looked at a total of 18 machines. Two forms of analysis were done on the hard drives of these machines.

- Raw Disk Walk
- File System Walk

### 5.1 Raw Disk Walk

The raw disk walk read in all the data on the disk and hashed it in 4 KB blocks at block boundaries. On analyzing this data, we observed the our estimates of commonality were skewed by the presence of unallocated garbage on the disk. This is why we modified this form of analysis to be file system specific and looked into the allocation tables for the allocation status of the blocks and recorded this information along with the hash. Currently this form of analysis is limited to the NTFS file system but can be extended to any other file system that allows us to access allocation information.

It should be noted that this approach does not destroy the OS agnostic nature of ISR. We only look at

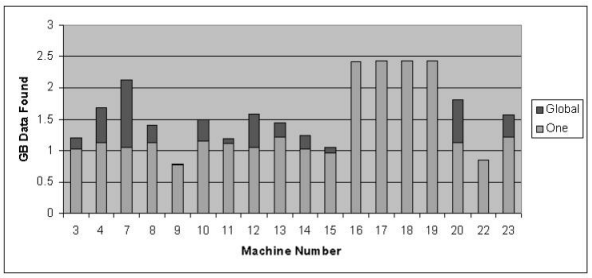


Figure 2: Block Commonality: GB of Unique Data Common

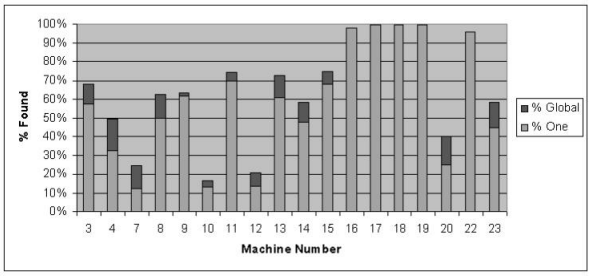


Figure 3: Block Commonality: Percentage of Unique Data found

allocation data to give an idea of the actual commonality that we can find. Our prototype system does not use any file system specific information. This absence of knowledge should not significantly affect the performance of the system when the disk is being reconstructed in parallel at runtime as all the blocks requested to be read by the OS will always be allocated. We could have a performance hit if the disk is rebuilt before the VM starts as the system would not know if it was fetching useful data or garbage. However it is possible that a working set algorithm could help is solving at least a part of this problem.

We examined 18 machines in total for this analysis out of which 8 machines were Windows XP while the rest 10 ran Windows 2000. Of of the 8 Windows XP machines, 5 of them were Virtual Machines that we used for our benchmarks. These 5 VMs are almost exact copies of each other and therefore not very important in our analysis. The rest of all the machines were actual machines in use by people at the Intel Research Lab, Pittsburgh and some students at Carnegie Mellon University. Out of the 10 Windows 2000 machines, 7 of them were machines built by the Intel IT department. While most of the IT machines have been heavily used 3 of them were built at the same time by ghosting of their hard drives and had been in use for less than two months. They thus exhibit a high degree of commonality. The question to be examined is whether this low modification ratio seen by us is typical of the common home or office user.

The results of the analysis are shown in Figures 2 and 3. All the machines presented in the graphs have been anonymized and given a number from 1 to 23. We however only had valid data from 18 of the machines for these results. In Figure 2 the lighter portion of the bars represents the maximum amount of data that can be found by looking at the best one machine out of the rest while the darker portion represents the increase we get from looking at the rest of the machines. Figure 3 shows us how what percentage of the disk can be reconstituted by the data that we found. As we can see, most of the people have one machine that they can find most of their data on.

## 5.2 File System Walk

Our second form of analysis was a file system walk where we hashed every file on disk and recorded the hash with the size. This was again only done on NTFS systems because we wanted to compare the data with the results from the allocated block analysis. Not all files present on disk could be hashed due to Windows locking some system files. The file analysis was later extended to give us the extensions and a limited depth view of directory structure for the directories containing system and application files. This was done to give us a better idea of where commonality could be found.

The list of machines examined by the File System Walk does not exactly match the list for the Disk Walk. This is because the analysis was at the discretion of the user and given the long time duration of the experiments, we had a few cases of people stopping the analysis in the middle. We also only analyzed one VM disk as we knew that the VM disk images were almost

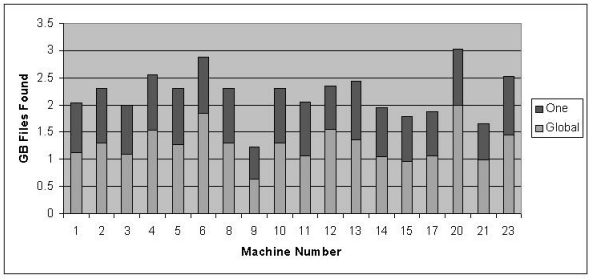


Figure 4: File Commonality: GB of Unique Data Common

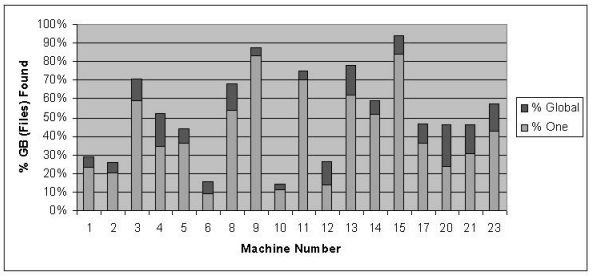


Figure 5: File Commonality: Percentage of Unique Data found

exact duplicates of each other.

The results of these experiments are presented in Figures 4 and 5. As earlier, the lighter portion of the bars represents the maximum amount of data that can be found by looking at the best one machine out of the rest while the darker portion represents the increase we get from looking at the rest of the machines.

## 5.3 Memory Savings

One of the problems that we observed while trying to achieve faster resume times is the fact that we need to load the file that encapsulates the RAM image of the VM. This file can be large and it is a little over 256 MB for the VMs we use for benchmarks. However, even this file could lend itself to the analysis described above as a large amount of memory will consist of pages from application files and the kernel. While it is hard to examine memory of a running machine, looking at a few of the suspended VMs memory files, we estimate that using a combination of the above Disk Decomposition techniques and gzip we could achieve a compression of 35-60%.

## 5.4 Analysis of Experiment Results

From our data, we can see that for commonality between any two computers of the same or similar builds, we expect that there will be a large percentage of OS binary files that will be common. For example any two installations of a common OS like Windows XP will share data in the range of 700 to 800 MB. Also, for common application suites, there will be a commonality of around 150 to 200 MB. As for user data files, they are probably the least likely to exhibit commonality with nearby machines although the situation might differ between collaborators with close proximity. Apart from collaborators, it is expected that user data files will most likely be demand paged in or prefetched. It should also be noted that we do not want to reconstruct the entire disk image before resume happens. In fact it is probably going to be a hybrid of disk reconstruction together with demand paging and prefetching.

For our initial prototype, we used a disk of around 4 GB with 256 MB of RAM. It can be argued that with current disk capacities, it is not uncommon to see disk sizes of 100 GB or more. However we believe that the working set of the user will be substantially smaller than the size of the disk. For example, if the bulk of the files occupying such a large disk are large read-only data files such as MP3s, Video clips, etc, they are usually not going to be accessed together and should instead by mounted into the VM as a network drive where they can be fetched on demand. Using a demand policy for all data not present in the working set would be very advantageous. We should also note that in the 100 GB scenario, it might be hard to find a high commonality percentage between two machines. However we need to remember that our main goal is to reduce resume latency and slowdown. Thus while as a raw number the commonality percentage might decrease for large hard drives, static commonality is not what we are interested in. What is more interesting is how much dynamic commonality exists, i.e. how

much of commonality can be found between the user's working set and the list of sources.

# 6 Weighted Commonality

As stated earlier, simply analyzing the contents of hard drives to look for commonality can be misleading. What we need to actually compare is a user's working set and the amount of this data from this set that can be served from nearby sources. For this purpose we have used two benchmarks that model user think time, concurrency, and switching between applications. To analyze weighted commonality, we have attached a jukebox of software to the Virtual Machine that serves as a source of data blocks.

While one of the goals of this system was to see performance across high latency networks, the benchmarks are flawed as they time out when a large amount of data is being fetched across a slow network. We are currently in the process of developing internal benchmarks that would replace the closed commercial ones.

## 6.1 Benchmarks

Both of the benchmarks are commercial benchmarks for Windows produced by the Business Applications Performance Corporation named SYSmark2002 Office Productivity (SOP) and SYSmark2002 Internet Content Creation (SICC).

SOP uses script-driven Windows applications to model a single user's office activity in an automobile company. The user creates and edits documents using Microsoft Word, Excel and Powerpoint. He accesses email using Microsoft Outlook and queries a database using Microsoft Access. He views presentations on the Web using Netscape Communicator. A part of the benchmark includes speech to text translation using Dragon NaturallySpeaking. Another part constructs a file archive using WinZip. McAfee VirusScan is run in the background during the entire benchmark.

Like SOP, SICC also consists of script-driven Windows applications. SICC models the work of a designer creating Web pages for a sports company using Macromedia Dreamweaver. He embeds images developed with Adobe Photoshop and animations created with Macromedia Flash. He then creates a promotional video using Adobe Premiere and encodes it using Windows Media Encoder.

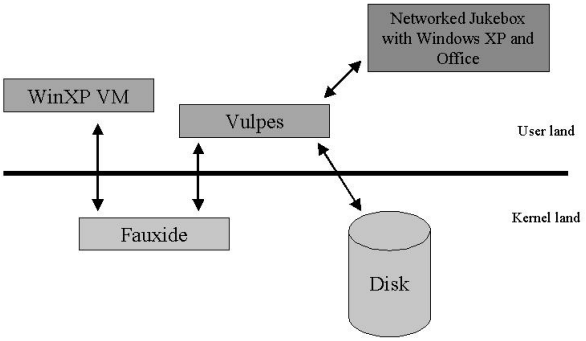


Figure 6: System architecture with Jukebox

## 6.2 Jukebox

As mentioned earlier, we wanted to look at the runtime commonality we could find by looking at the available sources of data. For this purpose, a jukebox was built that could serve up blocks of data based on a hash as the request. A diagram of the system architecture can be seen in Figure 6. In this diagram, *Fauxide* is our fake IDE disk that accept read or write requests while *Vulpes* is the corresponding user level process that does the translation from sector addresses to the corresponding file that we use to store that portion of the disk. While Vulpes would normally go out across the network in case it does a miss on the file, in this case it also knows the hash associated with the missing block. It therefore first asks the jukebox for data containing the hashed block. If it is found, it fetches the data from the jukebox, rehashes it to verify its contents and then passes it up to the VM. If the block is not found, it will be fetched from the DFS server. In case of writes to any particular block, the hash associated with the block is invalidated and the block is written back to the DFS. The invalidation is done to prevent fetching stale data from the jukebox in case the block is read again.

#### 6.2.1 Results from benchmarks

We failed to run the benchmarks across a low bandwidth link because of the fact that the scripts driving the benchmarks timed out. As the source to the benchmarks are not available to us, we were unable to fix this problem

To get an idea of commonality, we ran the benchmarks mentioned in Section 6.1 from local disk but connected a Jukebox to the system. The Jukebox was populated with blocks belonging to Windows XP and Office XP. While ultimately SHA-1 hashes are going to be used for the system, MD5 was used for this experiments. All requests for blocks were sent to the jukebox before it went to disk The block access readings were taken from start to end of benchmark and did not include the reboot the benchmarks do after finishing. The results of the data requests served are presented in Table 1. Note that this table only records unique requests, i.e. if a block with the same hash value is asked for twice, the request is not recorded again as once Vulpes has a block, it should not need to request it again.

Table 1: Data served by Jukebox

| Benchmark | Data Requested | Requests Served |
|---|---|---|
| SOP | 0.59 GB | 0.19 GB |
| SOP | 0.65 GB | 0.10 GB |

### 6.3 Analysis of Weighted Commonality

On further analysis, we discovered that if the Jukebox had all the software present on the Virtual Machines and none of the user data, we could serve up approximately 0.24 GB for SOP and 0.13 for SICC. This gives us a commonality of 40% for SOP and 20% for just applications.

These benchmarks are also slightly artificial with respect to both the amount of data read in 20 minutes and the amount of work accomplished. In real life we expect the commonality number and thus performance to be higher than these results. This expectation needs to be verified by further investigation. However even if these numbers are typical, a saving of 30-40% is still significant and should lead to an improvement in user response time.

## 7 Related Work

There has been a large amount of work in the content addressable storage space including Chord [17], CAN [15] and PAST [5]. Other people have also done a lot of analysis of machines hard disk and the most notable amongst these is the Farsite project [1]. Work on variable sized chunking such as the Low Bandwidth File System (LBFS) [11] and hash based network storage like Venti [14] has also influenced this work.

## 8 Future Work

One of the first things that needs to be done is to run the same set of hard drive analysis scripts on a bigger data set and to evaluate more operating systems. We also need to develop better benchmarks which work in low bandwidth scenarios so that we can evaluate the effect of content addressing on performance.

All the work described in this paper was done using fixed size blocks. It would however be interesting to see if variable sized blocks based on Rabin fingerprints such as those used in the LBFS system would give us better commonality.

While all the approaches described in this paper deal with blocks, a hybrid approach could also be advocated where we also look into the file system for hints on how to reconstruct the disk. This might help in proactively getting files to satisfy a block miss later. For example, suppose that we know that all versions of an OS will have a set of large binaries in common. It might then be useful to capture hashes of these files and proactively fetch it at resume time. All proactive fetches could then be kept in a local cache and the files hashed by 4K blocks similar to the way described earlier in Section 4. We would first check against these local hashes for any block miss before searching for it on the network. The hope is that this process could satisfy a number of misses as the VM runs.

It might also be interesting to see if a method could exist to intelligently deal with empty space on the vir-

tual disk. This might necessitate actually looking into the file system and therefore make this approach file system specific. However the general approach will always work with any system and so it might be useful to do this for commonly used file systems such as ext2, FAT and NTFS. Also, knowing which blocks were allocated would be very useful as only they could then be requested for and not unallocated garbage.

# 9 Acknowledgments

We would like to thank Yan Ke for his feedback on the ideas presented in this paper. All unidentified trademarks used in this paper belong to their respective owners.

# References

[1] W. J. Bolosky, S. Corbin, D. Goebel, and J. R. Douceur. Single instance storage in windows 2000.

[2] B. den Boer and A. Bosselaers. Collisions for the Compression Function of MD-5. *Lecture Notes in Computer Science*, 765:293–304, 1994.

[3] H. Dobbertin. The Status of MD5 After a Recent Attack. *CryptoBytes*, 1996.

[4] H. Dobbertin, A. Bosselaers, and B. Preneel. RIPEMD-160: A strengthened version of RIPEMD. In *Fast Software Encryption*, pages 71–82, 1996.

[5] P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. pages 75–80, 2001.

[6] L. Grinzo. Getting Virtual with VMWare 2.0. *Linux Magazine*, June 2000.

[7] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. In *Thirteenth ACM Symposium on Operating Systems Principles*, volume 25, pages 213–225, Asilomar Conference Center, Pacific Grove, U.S., 1991. ACM Press.

[8] M. Kozuch, M. Satyanarayanan, T. Bressoud, and Y. Ke. Efficient State Transfer for Internet Suspend/Resume. 2002.

[9] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*. ACM, November 2000.

[10] G. H. Kuenning and G. J. Popek. Automated hoarding for mobile computers. In *Symposium on Operating Systems Principles*, pages 264–275, 1997.

[11] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. In *Symposium on Operating Systems Principles*, pages 174–187, 2001.

[12] NIST. Secure Hash Standard (SHS). In *FIPS Publication 180-1*, 1995.

[13] J. Plank, M. Beck, W. Elwasif, T. Moore, M. Swany, and R. Wolski. The internet backplane protocol: Storage in the network, 1999.

[14] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *Proceedings of the FAST 2002 Conference on File and Storage Technologies*, 2002.

[15] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. Technical Report TR-00-010, Berkeley, CA, 2000.

[16] R. Rivest. The MD5 Message-Digest Algorithm. *RFC 1321*, 1992.

[17] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.